Session-Based Concurrency, Reactively

Mauricio Cano, Jaime Arias, Jorge A. Pérez

Session-based Concurrency, Reactively

Mauricio Cano[1], Jaime Arias[2], and Jorge A. Pérez[3]

[1] University of Groningen, The Netherlands
[2] Inria Grenoble Rhône-Alpes, France
[3] University of Groningen and CWI, Amsterdam, The Netherlands

Abstract. This paper concerns formal models for the analysis of communication-centric software systems that feature *declarative and reactive behaviors*. We focus on *session-based concurrency*, the interaction model induced by session types, which uses (variants of) the π-calculus as specification languages. While well-established, such process models are not expressive enough to specify declarative and reactive behaviors common in emerging communication-centric software systems. Here we propose the *synchronous reactive programming* paradigm as a uniform foundation for session-based concurrency. We present correct encodings of session-based calculi into ReactiveML, a synchronous reactive programming language. Our encodings bridge the gap between process specifications and concurrent programs in which session-based concurrency seamlessly coexists with declarative, reactive, timed, and contextual behaviors.

1 Introduction

In this paper, we introduce the *synchronous reactive programming* paradigm as a practical foundation for *communication-centric* software systems. Our motivation is twofold. First, synchronous reactive programming allows us to uniformly integrate point-to-point communications (as in the π-calculus) with declarative, reactive, timed, and contextual behaviors—this is an elusive combination for process models such as the π-calculus. Second, by relying on ReactiveML (a synchronous reactive programming language with a formal semantics), we may bridge the gap between π-calculus *processes* and actual concurrent *programs*, thus bringing a rigorous communication model to programmers.

Large software systems are deployed as aggregations of distributed interacting components, which are built using a myriad of different programming platforms and/or made available as black-boxes that expose minimal interaction interfaces. In these complex, heterogeneous systems *communication* emerges as the key unifying glue. Certifying that interacting components conform to their prescribed protocols is thus an important but challenging task, and is essential in ensuring overall system correctness.

Besides protocol conformance, analyzing communication-centric software systems entails addressing additional challenges, which can be seen as related to the increasing ubiquity of these systems. Indeed, communication-centric software appears in emerging trends (e.g., collective adaptive systems) and as such is subject to various classes of requirements that are orthogonal to communication correctness. We focus on communication-centric software systems featuring *declarative*, *reactive*, *timed*, and *contextual* behaviors. (In §2 we illustrate these intended systems, using a transactional protocol

subject to failures.) By stipulating governing conditions (rather than *how* to implement such conditions), declarative approaches naturally specify, e.g., security policies. Closely intertwined, constructs modeling reactivity, time, and context-awareness are at the heart of mechanisms that enforce, e.g., self-adaptation and fault-tolerance in dependable systems. Therefore, while not directly connected to protocol specifications, declarative, reactive, timed, and contextual behaviors (and their interplay) do influence communication and should be integrated into the analysis of protocol conformance.

Process calculi (such as the π-calculus [17]) have long offered a principled basis for the compositional analysis of message-passing programs. Within these approaches, our work concerns *session-based concurrency*, the interaction model induced by *session types* [11], which organize protocols as *sessions* between two or more participants. In session-based concurrency, a session type describes the contribution of each partner to the protocol. Interactions are structured, and always occur in matching pairs; e.g., when one partner sends, the other receives; when one partner offers a selection, the other chooses. Different session type theories for *binary* (two-party) and *multiparty* protocols have been developed [12]; here we focus on binary sessions.

Binary and multiparty session types rely on π-calculi with session constructs. These session calculi have been extended with declarative, reactive, timed, and contextual behaviors, but none of these extensions captures all these features. For instance, session calculi with *assertions* (logical predicates) [5,3] may describe certain declarative requirements, but do not account for reactive and contextual behaviors. Frameworks with time-related conditions, such as [4,1], have similar limitations. The framework in [13] supports contextual information through *events*, but does not represent reactive, declarative behaviors. Integrating these extensions into a single process framework seems rather difficult, for they rely on different languages and often conflicting assumptions.

Here we pursue a different approach: we embed session-based concurrency within the synchronous reactive programming (SRP) model for reactive, timed systems [2,10]. Hence, rather than extending session π-calculi with declarative, reactive, timed, and contextual features, we encode session-based communication into a setting where these features (and their interplay) are already well understood. We consider ReactiveML, a programming language based on SRP [16,15], as target language in our developments. ReactiveML is a general purpose functional language with a well-defined formal semantics. Our ***technical contributions*** are two correct encodings of session π-calculi into ReactiveML. In a nutshell, we use *signals* in ReactiveML to mimick *names* in session π-calculi. Our encodings enable us to integrate, in a seamless and uniform way, session-based constructs as "macros" in ReactiveML programs with declarative and reactive constructs. Moreover, since our encodings are executable (well-typed) ReactiveML programs, our results have a direct practical character, which serves to bridge the gap between specifications in process models and actual concurrent programs.

This paper is structured as follows. §2 illustrates our approach via an example. §3 summarizes the syntax and semantics of a session π-calculus and of ReactiveML. In both cases, we consider languages with synchronous and asynchronous (queue-based) communication. §4 presents our two encodings and states their correctness. §5 collects closing remarks. An online appendix includes further examples and technical details (omitted definitions and proofs) [7].

2 A Motivating Example

We use a toy example to illustrate (i) the limitations of session π-calculi in representing structured communications with declarative/reactive behaviors, and (ii) how our approach, based on encodings into ReactiveML, can neatly overcome such limitations.

A Ride Protocol Suppose a conference attendee who finds himself in a foreign airport. To get in time for his presentation, he uses a mobile app in his phone to request a ride to the conference venue. The intended protocol may be intuitively described as follows:

1. *Attendee* sends his current location and destination to a neighbouring *Driver*.
2. *Driver* receives these two pieces of information and offers three options to *Attendee*: a ride right *now*, a ride at a *later* time, or to *abort* the transaction.
3. *Attendee* is in a hurry, and so he selects to be picked up right now.
4. *Driver* replies by sending an estimated arrival time at *Attendee*'s location.

Using session π-calculus processes (as in, e.g., [18]), this protocol may be implemented as a process $S = (\boldsymbol{\nu} xy)(A(x) \mid D(y))$, where processes $A(x)$ and $D(y)$, abstract the behavior of *Attendee* and *Driver* as follows:

$$A(x) = x\langle loc\rangle.x\langle des\rangle.x \triangleleft \mathsf{now}.x(e).\mathbf{0}$$
$$D(y) = y(l).y(d).y \triangleright \{\mathsf{now} : y\langle eta\rangle.\mathbf{0}\,, \mathsf{later} : y(t).y\langle ok\rangle.\mathbf{0}\,, \mathsf{quit} : Close_y\}$$

where process $Close_y$ denotes an unspecified sub-protocol for closing the transaction. Above, we write $x\langle z\rangle.P$ (resp. $x(w).P$) to denote the output (resp. input) along name x with continuation P. Processes $x \triangleleft l.P$ and $x \triangleright \{l_i : P_i\}_{i\in I}$ denote internal and external labeled choices, respectively. Above, now, later, and quit denote labels. Process $\mathbf{0}$ denotes inaction. Process $(\boldsymbol{\nu} xy)P$ declares x and y as dual *session endpoints* in P. This way, S says that $A(x)$ and $D(y)$ play complementary roles in the session protocol.

The Need for Richer Behaviors Session-based concurrency assumes that once a session is established, communication may proceed without interruptions. This is unrealistic in most real-life scenarios, where established sessions are prone to failures or interruptions. For instance, a connectivity issue in the middle of the protocol with *Driver* may leave *Attendee* stuck in the airport. In such cases, notions of *contextual information*, *reactivity*, and *time* become essential:

Contextual Information such as, e.g., external events signalling a malfunction, allows relating the system with its environment. For instance, we may like to relate $A(x)$ and $D(y)$ with a connectivity manager that triggers warning events.

Reactivity serves to detect unforeseen circumstances (e.g., failures) and to define appropriate system behaviors to run in such cases. For instance, we may like to define $A(x)$ so that another driver is requested if a failure in a protocol with $D(y)$ arises.

Time allows to track the instant in which a failure occurred, and also to establish a deadline within which the failure should be resolved. For instance, in case of failure $A(x)$ may try contacting alternative drivers only until k instants after the failure.

As mentioned above, the session π-calculus does not support these features, and proposed extensions do not comprehensively address them. We rely on synchronous reactive programming (SRP) and ReactiveML, which already have the ingredients for seamlessly integrating declarative, reactive behavior into session-based concurrency.

ReactiveML ReactiveML extends OCaml with reactive, timed behavior. Time is modelled as discrete units, called *instants*; reactivity arises through *signals*, which may carry values. In ReactiveML, expression $\mathsf{signal}\ x\ \mathsf{in}\ e$ declares a new signal x. We use constructs $\mathsf{emit}\ s\ v$ and $\mathsf{await}\ s(x)\ \mathsf{in}\ e$ to emit and await a signal s, respectively. Preemption based on signals is obtained by the expression $\mathsf{do}\ (e_1)\ \mathsf{until}\ s \rightarrow (e_2)$, which executes e_1 until signal s is detected, and runs e_2 in the next instant. Moreover, ReactiveML can encode the parallel composition of expressions e_1 and e_2, denoted $e_1 \parallel e_2$.

Embedding Sessions in ReactiveML Our first encoding, denoted $[\![\cdot]\!]_f$ (cf. Def. 14), translates session π-calculus processes into ReactiveML expressions; we use substitution f to represent names in the session π-calculus using (fresh) signals in ReactiveML. Our second encoding, denoted $(\!|\cdot|\!)$ (cf. Def. 17), supports an asynchronous semantics.

We illustrate $[\![\cdot]\!]_f$ by revisiting our example above. Let us define a concurrent reactive program in which $[\![A(x)]\!]_f$, $[\![D(y)]\!]_f$, and $[\![D'(w)]\!]_f$ represent ReactiveML snippets that implement session-based communication. We consider a simple possibility for failure: that *Driver* ($D(y)$) may cancel a ride anytime or that communication with *Attendee* ($A(x)$) fails and cannot be recovered. Ideally, we would like a new driver $D'(w)$, whose implementation may be the same as $D(y)$, to continue with the protocol, without disrupting the protocol from the perspective of $A(x)$. This could be easily expressed in ReactiveML as the expression $S' = \mathsf{signal}\ w_1, w_2\ \mathsf{in}\ (RA \parallel RD)$ where:

$$
\begin{array}{l}
RA = \mathsf{do}\ ([\![A(x)]\!]_{\{x \leftarrow w_1\}})\ \mathsf{until}\ fail \rightarrow (\mathsf{await}\ w_2(z)\ \mathsf{in}\ [\![A(x)]\!]_{\{x \leftarrow z\}}) \\
RD = \mathsf{do}\ ([\![D(y)]\!]_{\{y \leftarrow w_1\}})\ \mathsf{until}\ fail \rightarrow (BD) \\
BD = \mathsf{signal}\ w_3\ \mathsf{in}\ (\mathsf{emit}\ w_2\ w_3; [\![D'(w)]\!]_{\{w \leftarrow w_3\}})
\end{array}
$$

S' declares two signals: while signal w_1 connects a reactive attendee RA and the reactive driver RD, signal w_2 connects RA with a backup driver BD. If no failure arises, RA and RD run their expected session protocol. Otherwise, the presence of signal $fail$ will be detected by both RA and RD: as a result, the attendee will await a new signal for restarting the session; process $[\![D(y)]\!]$ stops and BD will become active in the next instant. After emitting a fresh signal w_3, BD can execute the protocol with RA.

3 Preliminaries

A Session π-calculus Our presentation follows closely that of [18]. We assume a countable infinite set of variables $\mathcal{V}_s$, ranged over by $x, y, \ldots$. A variable represents one of the two *endpoints* of a session. We use $v, v', \ldots$ to range over *values*, which include variables and the boolean constants $\mathtt{tt}, \mathtt{ff}$. Also, we use $l, l', \ldots$ to range over *labels*. We write $\widetilde{x}$ to denote a finite sequence of variables (and similarly for other elements).

Definition 1 (π). *The set π of session processes is defined as:*

$$
\begin{array}{rl}
P, Q ::= & x\langle v\rangle.P \mid x(y).P \mid x \triangleleft l.P \mid x \triangleright \{l_i : P_i\}_{i \in I} \mid v?\,(P) \mathbf{:} (Q) \mid P \mid Q \mid \mathbf{0} \\
& \mid (\boldsymbol{\nu} xy)P \mid *x(y).P
\end{array}
$$

Process $x\langle v\rangle.P$ sends value v over x and then continues as P; dually, process $x(y).Q$ expects a value v on x that will replace all free occurrences of y in Q. Processes $x \triangleleft l_j.P$ and $x \triangleright \{l_i : Q_i\}_{i \in I}$ define a labeled choice mechanism, with labels indexed by the finite set I: given $j \in I$, process $x \triangleleft l_j.P$ uses x to select l_j and trigger process Q_j.

$$\lfloor \textsc{Com} \rfloor \;\; (\boldsymbol{\nu} xy)(x\langle v\rangle.P \mid y(z).Q) \longrightarrow (\boldsymbol{\nu} xy)(P \mid Q\{^{v}/_{z}\})$$
$$\lfloor \textsc{Sel} \rfloor \;\; (\boldsymbol{\nu} xy)(x \triangleleft l_j.P \mid y \triangleright \{l_i{:}Q_i\}_{i\in I}) \longrightarrow (\boldsymbol{\nu} xy)(P \mid Q_j)\ (j \in I)$$
$$\lfloor \textsc{Rep} \rfloor \;\; (\boldsymbol{\nu} xy)(x\langle v\rangle.P \mid *\, y(z).Q) \longrightarrow (\boldsymbol{\nu} xy)(P \mid Q\{^{v}/_{z}\} \mid *\, y(z).Q)$$
$$\lfloor \textsc{IfT} \rfloor \;\; \mathtt{tt}?\,(P)\mathtt{:}(Q) \longrightarrow P \qquad \lfloor \textsc{IfF} \rfloor \;\; \mathtt{ff}?\,(P)\mathtt{:}(Q) \longrightarrow Q$$

Fig. 1. Reduction relation for π processes (contextual congruence rules omitted).

We assume pairwise distinct labels. The conditional process $v?\,(P)\,\mathtt{:}\,(Q)$ behaves as P if v evaluates to $\mathtt{tt}$; otherwise it behaves as Q. Parallel composition and inaction are standard. We often write $\prod_{i=1}^{n} P_i$ to stand for $P_1 \mid \cdots \mid P_n$. The *double restriction* $(\boldsymbol{\nu} xy)P$ binds together x and y in P, thus indicating that they are the two endpoints of a session. Process $*\,x(y).P$ denotes a replicated input process, which allows us to express infinite server behaviors. In $x(y).P$ (resp. $(\boldsymbol{\nu} yz)P$) occurrences of y (resp. y, z) are bound with scope P. The set of free variables of P, denoted $\mathsf{fv}(P)$, is as expected.

The operational semantics for π is given as a *reduction relation* $\longrightarrow$, the smallest relation generated by the rules in Fig. 1. Reduction expresses the computation steps that a process performs on its own. It relies on a *structural congruence* on processes, denoted $\equiv_{\mathsf{S}}$, which identifies processes up to consistent renaming of bound variables, denoted $\equiv_{\alpha}$. Formally, $\equiv_{\mathsf{S}}$ is the smallest congruence that satisfies the axioms:

$$P \mid \mathbf{0} \equiv_{\mathsf{S}} P \quad P \mid Q \equiv_{\mathsf{S}} Q \mid P \quad P \equiv_{\mathsf{S}} Q \text{ if } P \equiv_{\alpha} Q$$
$$(P \mid Q) \mid R \equiv_{\mathsf{S}} P \mid (Q \mid R) \quad (\boldsymbol{\nu} xy)(\boldsymbol{\nu} wz)P \equiv_{\mathsf{S}} (\boldsymbol{\nu} wz)(\boldsymbol{\nu} xy)P$$
$$(\boldsymbol{\nu} xy)\mathbf{0} \equiv_{\mathsf{S}} \mathbf{0} \quad (\boldsymbol{\nu} xy)P \mid Q \equiv_{\mathsf{S}} (\boldsymbol{\nu} xy)(P \mid Q) \quad \text{if } x, y \notin \mathsf{fv}(Q)$$

We briefly comment on the rules in Fig. 1. Reduction requires an enclosing restriction $(\boldsymbol{\nu} xy)(\cdots)$; this represents the fact that a session connecting endpoints x and y has been already established. Rule $\lfloor \textsc{Com} \rfloor$ represents the synchronous communication of value v through endpoint x to endpoint y. While Rule $\lfloor \textsc{Sel} \rfloor$ formalizes a labeled choice mechanism, in which communication of a label l_j is used to choose which of the Q_i will be executed, Rule $\lfloor \textsc{Repl} \rfloor$ is similar to Rule $\lfloor \textsc{Com} \rfloor$, and used to spawn a new copy of Q, available as a replicated server. Rules $\lfloor \textsc{IfT} \rfloor$ and $\lfloor \textsc{IfF} \rfloor$ are self-explanatory. Rules for reduction within parallel, restriction, and $\equiv_{\mathsf{S}}$ (not given in Fig. 1) are as expected.

The following notion will be useful in stating properties of our translations.

Definition 2 (Contexts for π). *The syntax of (evaluation) contexts in π is given by the following grammar: $E ::= [\cdot] \mid E \mid P \mid P \mid E \mid (\boldsymbol{\nu} xy)(E)$, where P is a π process and '$[\cdot]$' represents a 'hole'. We write $C[\cdot]$ to range over contexts $(\boldsymbol{\nu}\widetilde{xy})([\cdot] \mid P_1 \mid \ldots \mid P_n)$, with $n \geq 1$. $E[P]$ (resp. $C[P]$) will denote the process obtained by filling $[\cdot]$ with P.*

An Asynchronous Session π-calculus ($\mathsf{a}\pi$) Following [13], we now define $\mathsf{a}\pi$, a variant of π with asynchronous (queue-based) semantics. The syntax of $\mathsf{a}\pi$ includes variables $x, y, \ldots$ and *co-variables*, denoted $\overline{x}, \overline{y}$. Intuitively, x and $\overline{x}$ denote the two endpoints of a session, with $\overline{\overline{x}} = x$. We write $\mathcal{V}_a$ to denote the set of variables and co-variables; k, k' will be used to range over $\mathcal{V}_a$. As before, values include booleans and variables. The syntax of processes is as follows:

$$\lfloor \textsc{Send} \rfloor \;\; x\langle v\rangle.P \mid x[i:\widetilde{m_1}, o:\widetilde{m_2}] \longrightarrow_{\mathtt{A}} P \mid x[i:\widetilde{m_1}, o:\widetilde{m_2}\cdot v]$$
$$\lfloor \textsc{Sel} \rfloor \;\; x \triangleleft l.P \mid x[i:\widetilde{m_1}, o:\widetilde{m_2}] \longrightarrow_{\mathtt{A}} P \mid x[i:\widetilde{m_1}, o:\widetilde{m_2}\cdot l]$$
$$\lfloor \textsc{Com} \rfloor \;\; x[i:\widetilde{m_1}, o:m\cdot\widetilde{m_2}] \mid \overline{x}[i:\widetilde{m_1}, o:\widetilde{m_2}] \longrightarrow_{\mathtt{A}} x[i:\widetilde{m_1}, o:\widetilde{m_2}] \mid \overline{x}[i:\widetilde{m_1}\cdot m, o:\widetilde{m_2}]$$
$$\lfloor \textsc{Recv} \rfloor \;\; x(y).P \mid x[i:v\cdot\widetilde{m_1}, o:\widetilde{m_2}] \longrightarrow_{\mathtt{A}} P\{v/y\} \mid x[i:\widetilde{m_1}, o:\widetilde{m_2}]$$
$$\lfloor \textsc{Bra} \rfloor \;\; x \triangleright \{l_i : P_i\}_{i\in I} \mid x[i:l_j\cdot\widetilde{m_1}, o:\widetilde{m_2}] \longrightarrow_{\mathtt{A}} P_j \mid x[i:\widetilde{m_1}, o:\widetilde{m_2}] \quad (j\in I)$$
$$\lfloor \textsc{IfT} \rfloor \;\; \mathtt{tt}?\,(P)\mathtt{:}(Q) \longrightarrow_{\mathtt{A}} P \qquad \lfloor \textsc{IfF} \rfloor \;\; \mathtt{ff}?\,(P)\mathtt{:}(Q) \longrightarrow_{\mathtt{A}} Q$$

Fig. 2. Reduction relation for $\mathsf{a}\pi$ processes (contextual congruence rules omitted).

Definition 3 ($\mathsf{a}\pi$ and $\mathsf{a}\pi^\star$). *The set $\mathsf{a}\pi$ of asynchronous session processes is defined as:*

$$\begin{array}{rl} P,Q ::= & k\langle v\rangle.P \mid k(y).P \mid k\triangleleft l.P \mid k\triangleright\{l_i:P_i\}_{i\in I} \mid v?\,(P)\mathtt{:}(Q) \mid P\mid Q \mid \mathbf{0} \\ & \mid \;\; (\boldsymbol{\nu}x)P \mid \mu X.P \mid X \mid k[i:\widetilde{m}; o:\widetilde{m}] \end{array}$$

We write $\mathsf{a}\pi^\star$ to denote the sub-language of $\mathsf{a}\pi$ without queues.

Differences with respect to Def. 1 appear in the second line of the above grammar. The usual (single) restriction $(\boldsymbol{\nu}x)P$ is convenient in a queue-based setting; it binds both x and $\overline{x}$ in P. We consider recursion $\mu X.P$ rather than input-guarded replication. Communication in $\mathsf{a}\pi$ is mediated by queues of messages m (values v or labels l), one for each endpoint k; these queues, denoted $k[i:\widetilde{m}; o:\widetilde{m}]$, have output and input parts. Synchronization proceeds as follows: the sending endpoint first enqueues the message m in its own output queue; then, m is moved to the input queue of the receiving endpoint; finally, the receiving endpoint retrieves m from its input queue. We will use ϵ to denote the empty queue. Notions of free/bound (recursive) variables are as expected.

The operational semantics of $\mathsf{a}\pi$ is defined as a reduction relation coupled with a structural congruence relation $\equiv_{\mathtt{A}}$. The former is defined by the rules in Fig. 2, which either follow the above intuitions for queue-based message passing or are exactly as for π; the latter is defined as the smallest congruence on processes that considers standard principles for parallel composition and inaction, together with the axioms:

$$(\boldsymbol{\nu}x)(\boldsymbol{\nu}y)P \equiv_{\mathtt{A}} (\boldsymbol{\nu}y)(\boldsymbol{\nu}x)P \qquad (\boldsymbol{\nu}x)\mathbf{0} \equiv_{\mathtt{A}} \mathbf{0} \quad \mu X.P \equiv_{\mathtt{A}} P\{\mu X.P/X\}$$
$$k[i:\epsilon; o:\epsilon] \equiv_{\mathtt{A}} \mathbf{0} \qquad (\boldsymbol{\nu}x)P \mid Q \equiv_{\mathtt{A}} (\boldsymbol{\nu}x)(P\mid Q) \text{ if } x\notin \mathsf{fv}(Q).$$

The notion of contexts for $\mathsf{a}\pi$ includes unary contexts E and binary contexts C:

Definition 4 (Contexts for $\mathsf{a}\pi$). *The syntax of contexts in $\mathsf{a}\pi$ is given by the following grammar: $E ::= [\cdot] \mid E \mid P \mid P \mid E \mid (\boldsymbol{\nu}x)E$, where P is an $\mathsf{a}\pi$ process and '$[\cdot]$' represents a 'hole'. We write $C[\cdot_1,\cdot_2]$ to denote binary contexts $(\boldsymbol{\nu}\widetilde{x})([\cdot_1]\mid[\cdot_2]\mid\prod_{i=1}^n P_i)$ with $n\geq 1$. We will write $E[P]$ (resp. $C[P,Q]$) to denote the $\mathsf{a}\pi$ process obtained by filling the hole in $E[\cdot]$ (resp. $C[\cdot_1,\cdot_2]$) with P (resp. P and Q).*

Both π and $\mathsf{a}\pi$ abstract from an explicit phase of session initiation in which endpoints are bound together. We thus find it useful to identify $\mathsf{a}\pi$ processes which are *properly initialized (PI)*: intuitively, processes that contain all queues required to reduce.

Definition 5 (Properly Initialized Processes). *Let $P\equiv_{\mathtt{A}}(\boldsymbol{\nu}\widetilde{x})(P_1\mid P_2)$ be an $\mathsf{a}\pi$ process such that P_1 is in $\mathsf{a}\pi^\star$ (i.e., it does not include queues) and $\mathsf{fv}(P_1)=\{k_1,\ldots,k_n\}$. We say P is* properly initialized (PI) *if P_2 contains a queue for each session declared in P_1, i.e., if $P_2 = k_1[i:\epsilon,o:\epsilon]\mid\cdots\mid k_n[i:\epsilon,o:\epsilon]$.*

ReactiveML: A synchronous reactive programming language Based on the reactive model given in [6], ReactiveML [16] is an extension of OCaml that allows unbounded time response from processes, avoiding causality issues present in other SRP approaches. ReactiveML extends OCaml with *processes*: state machines whose behavior can be executed through several *instants*. Processes are the reactive counterpart of OCaml functions, which ReactiveML executes instantaneously. In ReactiveML, synchronization is based on *signals*: events that occur in one instant. Signals can trigger reactions in processes; these reactions can be run instantaneously or in the next instant. Signals carry values and can be emitted from different processes in the same instant.

We present the syntax of ReactiveML following [14], together with two semantics, with synchronous and asynchronous communication. We will assume countable infinite sets of variables $\mathcal{V}_r$ and names $\mathcal{N}_r$ (ranged over by x_1, x_2 and n_1, n_2, respectively).

Definition 6 (RML)**.** *The set* RML *of ReactiveML expressions is defined as:*

$$
\begin{array}{rcl}
v, v' & ::= & c \mid (v, v) \mid n \mid \lambda x.e \mid \mathsf{process}\ e \\
e, e' & ::= & x \mid c \mid (e, e) \mid \lambda x.e \mid e\ e \mid \mathsf{rec}\ x = v \\
& \mid & \mathsf{match}\ e\ \mathsf{with}\ \{c_i \to e_i\}_{i \in I} \mid \mathsf{let}\ x = e\ \mathsf{and}\ x = e\ \mathsf{in}\ e \mid \mathsf{run}\ e \mid \mathsf{loop}\ e \\
& \mid & \mathsf{signal}_e\ x : e\ \mathsf{in}\ e \mid \mathsf{emit}\ e\ e \mid \mathsf{pause} \mid \mathsf{process}\ e \\
& \mid & \mathsf{present}\ e?\,(e) : e \mid \mathsf{do}\ e\ \mathsf{when}\ e \mid \mathsf{do}\ (e)\ \mathsf{until}\ e(x) \to (e)
\end{array}
$$

Values $v, v', \ldots$ include constants c (booleans and the unit value $()$), pairs, names, abstractions, and also processes, which are made of expressions. The syntax of expressions e, e' extends a standard functional substrate with match and let expressions and with process- and signal-related constructs. Expressions $\mathsf{run}\ e$ and $\mathsf{loop}\ e$ follow the expected intuitions. Expression $\mathsf{signal}_g\ x : d\ \mathsf{in}\ e$ declares a signal x with default value d, bound in e; here g denotes a *gathering function* that collects the values produced by x in one instant. When d and g are unimportant (e.g., when the signal will only be emitted once), we will write simply $\mathsf{signal}\ x\ \mathsf{in}\ P$. We will also write $\mathsf{signal}\ x_1, \ldots, x_n\ \mathsf{in}\ e$ when declaring $n > 1$ distinct signals in e. If expression e_1 transitions to the name of a signal then $\mathsf{emit}\ e_1\ e_2$ emits a signal carrying the value from the instantaneous execution of e_2. Expression pause postpones execution to the next instant. The conditional expression $\mathsf{present}\ e_1?\,(e_2) : (e_3)$ checks the presence of a signal: if e_1 transitions to the name of a signal present in the current instant, then e_2 is run in the same instant; otherwise, e_3 is run in the next instant. Expression $\mathsf{do}\ e\ \mathsf{when}\ e_1$ executes e only when e_1 transitions to the name of a signal present in the current instant, and suspends its execution otherwise. Expression $\mathsf{do}\ (e_1)\ \mathsf{until}\ e(x) \to (e_2)$ executes e_1 until e transitions into the name of a signal currently present that carries a value which will substitute x. If this occurs, the execution of e_1 stops at the end of the instant and e_2 is executed in the next one. Using these basic constructs, we may obtain the useful derived expressions reported in Fig. 3, which include the parallel composition $e_1 \parallel e_2$. We will say that an expression with no parallel composition operator at top level is a *thread*.

We write $\equiv_{\mathtt{R}}$ to denote the smallest equivalence that satisfies the following axioms: (i) $e \parallel () \equiv_{\mathtt{R}} e$; (ii) $e_1 \parallel e_2 \equiv_{\mathtt{R}} e_2 \parallel e_1$; (iii) $(e_1 \parallel e_2) \parallel e_3 \equiv_{\mathtt{R}} e_1 \parallel (e_2 \parallel e_3)$.

A Synchronous Semantics for RML Following [14], we define a big-step operational semantics for RML. We require some auxiliary definitions for *signal environments* and *events*. Below, $\uplus$ and $\sqsubseteq$ denote usual multiset union and inclusion, respectively.

$$e_1 \parallel e_2 \triangleq \mathsf{let}\ \_ = e_1\ \mathsf{and}\ \_ = e_2\ \mathsf{in}\ () \qquad e_1; e_2 \triangleq \mathsf{let}\ \_ = ()\ \mathsf{and}\ \_ = e_1\ \mathsf{in}\ e_2$$
$$\mathsf{await}\ e_1(x)\ \mathsf{in}\ e_2 \triangleq \mathsf{do}\ (\mathsf{loop}\ \mathsf{pause})\ \mathsf{until}\ e_1(x) \to (e_2)$$
$$\mathsf{let\ rec\ process}\ f\ x_1 \ldots x_n = e_1\ \mathsf{in}\ e_2 \triangleq \mathsf{let}\ f = (\mathsf{rec}\ f = \lambda x_1 \ldots x_n.\mathsf{process}\ e_1)\ \mathsf{in}\ e_2 \ \ (n \geq 1)$$
$$\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \triangleq \mathsf{match}\ e_1\ \mathsf{with}\ \{\mathtt{tt} \to e_2 \,|\, \mathtt{ff} \to e_3\}$$

Fig. 3. Derived RML expressions.

Definition 7 (Signal Environment). *Let $\mathcal{D}$, $\mathcal{G}$, $\mathcal{M}$ be sets of default values, gathering functions, and multisets, respectively. A* signal environment *is a function $S : \mathcal{N}_r \to (\mathcal{D} \times \mathcal{G} \times \mathcal{M})$, denoted $S \triangleq [(d_1, g_1, m_1)/n_1, \ldots, (d_k, g_k, m_k)/n_k]$, with $k \geq 1$.*

We use the following notations: $S^d(n_i) = d_i$, $S^g(n_i) = g_i$, and $S^m(n_i) = m_i$. Also, $S^v = \mathit{fold}\ g_i\ m_i\ d_i$ where *fold* recursively gathers multiple emissions of different values in the same signal; see [16,14] for details. An event E associates a signal n_i to a multiset m_i that represents the values emitted during an instant:

Definition 8 (Events). *An* event *is defined as a function $E : \mathcal{N}_r \to \mathcal{M}$, i.e., $E \triangleq [m_1/n_1, \ldots, m_k/n_k]$, with $k \geq 1$. Given events E_1 and E_2, we say that E_1 is* included *in E_2 (written $E_1 \sqsubseteq_{\mathsf{E}} E_2$) if and only if $\forall n \in Dom(E_1) \cup Dom(E_2) \Rightarrow E_1(n) \sqsubseteq E_2(n)$. The* union *$E_1$ and E_2 (written $E_1 \sqcup_{\mathsf{E}} E_2$) is defined for all $n \in Dom(E_1) \cup Dom(E_2)$ as $(E_1 \sqcup_{\mathsf{E}} E_2)(n) = E_1(n) \uplus E_2(n)$.*

We now define the semantics of RML expressions. A big-step transition in RML captures reactions within a single instant, and is of the form $e \xrightarrow[S]{E,b} e'$ where S stands for the smallest signal environment (wrt $\sqsubseteq_{\mathsf{E}}$ and S^m) containing input, output, and local signals; E is the event made of signals emitted during the reaction; $b \in \{\mathtt{tt}, \mathtt{ff}\}$ is a boolean value that indicates termination: b is false if e is stuck during that instant and is true otherwise. At each instant i, the program reads an input I_i and produces an output O_i. The reaction of an expression obeys four conditions: (C1) $(I_i \sqcup_{\mathsf{E}} E_i) \sqsubseteq_{\mathsf{E}} S_i^m$ (i.e., S must contain the inputs and emitted signals); (C2) $O_i \sqsubseteq_{\mathsf{E}} E_i$ (i.e., the output signals are included in the emitted signals); (C3) $S_i^d \subseteq S_{i+1}^d$; and (C4) $S_i^g \subseteq S_{i+1}^g$ (i.e., default values and gathering functions are preserved throughout instants).

Fig. 4 gives selected transition rules; see [7] for a full account. Rules $\lfloor$L-PAR$\rfloor$ and $\lfloor$L-DONE$\rfloor$ handle let expressions, distinguishing when (a) at least one of the parallel branches has not yet terminated, and (b) both branches have terminated and their resulting values can be used. Rule $\lfloor$RUN$\rfloor$ ensures that declared processes can only be executed while they are preceded by run . Rules $\lfloor$LP-STU$\rfloor$ and $\lfloor$LP-UN$\rfloor$ handle loop expressions: the former decrees that a loop will stop executing when the termination boolean of its body becomes ff; the latter executes a loop until Rule $\lfloor$LP-STU$\rfloor$ is applied. Rule $\lfloor$SIG-DEC$\rfloor$ declares a signal by instantiating it with a fresh name in the continuation; its default value and gathering function must be instantaneous expressions. Rule $\lfloor$EMIT$\rfloor$ governs signal emission. Rule $\lfloor$PAUSE$\rfloor$ suspends the process for an instant. Rules $\lfloor$SIG-P$\rfloor$ and $\lfloor$SIG-NP$\rfloor$ check for presence of a signal n: when n is currently present, the body e_2 is run in the same instant; otherwise, e_3 is executed in the next instant. Rules $\lfloor$DU-END$\rfloor$, $\lfloor$DU-P$\rfloor$, and $\lfloor$DU-NP$\rfloor$ handle expressions do (e_1) until $e_2(x) \to (e_3)$. Rule $\lfloor$DU-END$\rfloor$ says that if e_1 terminates instanta-

$$\lfloor\text{L-PAR}\rfloor\ \frac{e_1 \xrightarrow[S]{E_1,b_1} e_1' \quad e_2 \xrightarrow[S]{E_2,b_2} e_2' \quad b_1 \wedge b_2 = \texttt{ff}}{\mathsf{let}\ x_1 = e_1\ \mathsf{and}\ x_2 = e_2\ \mathsf{in}\ e_3 \xrightarrow[S]{E_1 \sqcup_{\mathsf{E}} E_2,\texttt{ff}} \mathsf{let}\ x_1 = e_1'\ \mathsf{and}\ x_2 = e_2'\ \mathsf{in}\ e_3}$$

$$\lfloor\text{L-DONE}\rfloor\ \frac{e_1 \xrightarrow[S]{E_1,\texttt{tt}} v_1 \quad e_2 \xrightarrow[S]{E_2,\texttt{tt}} v_2 \quad e_3\{v_1, v_2/x_1, x_2\} \xrightarrow[S]{E_3,b} e_3'}{\mathsf{let}\ x_1 = e_1\ \mathsf{and}\ x_2 = e_2\ \mathsf{in}\ e_3 \xrightarrow[S]{E_1 \sqcup_{\mathsf{E}} E_2 \sqcup_{\mathsf{E}} E_3,b} e_3'}$$

$$\lfloor\text{RUN}\rfloor\ \frac{e \xrightarrow[S]{E_1,\texttt{tt}} \mathsf{process}\ e' \quad e' \xrightarrow[S]{E_2,b} e''}{\mathsf{run}\ e \xrightarrow[S]{E_1 \sqcup_{\mathsf{E}} E_2,b} e''} \qquad \lfloor\text{LP-STU}\rfloor\ \frac{e \xrightarrow[S]{E,\texttt{ff}} e'}{\mathsf{loop}\ e \xrightarrow[S]{E,\texttt{ff}} e'; \mathsf{loop}\ e} \qquad \lfloor\text{LP-UN}\rfloor\ \frac{e \xrightarrow[S]{E_1,\texttt{tt}} v \quad \mathsf{loop}\ e \xrightarrow[S]{E_2,b} e'}{\mathsf{loop}\ e \xrightarrow[S]{E_1 \sqcup_{\mathsf{E}} E_2,b} e'}$$

$$\lfloor\text{SIG-DEC}\rfloor\ \frac{e_1 \xrightarrow[S]{E_1,\texttt{tt}} v_1 \quad e_2 \xrightarrow[S]{E_2,\texttt{tt}} v_2 \quad e_3\{n/x\} \xrightarrow[S]{E_3,b} e_3' \quad n\ \text{fresh} \quad S(n) = (v_1, v_2, m)}{\mathsf{signal}_{e_2}\ x : e_1\ \mathsf{in}\ e_3 \xrightarrow[S]{E_1 \sqcup_{\mathsf{E}} E_2 \sqcup_{\mathsf{E}} E_3,b} e_3'}$$

$$\lfloor\text{EMIT}\rfloor\ \frac{e_1 \xrightarrow[S]{E_1,\texttt{tt}} n \quad e_2 \xrightarrow[S]{E_2,\texttt{tt}} v}{\mathsf{emit}\ e_1\ e_2 \xrightarrow[S]{E_1 \sqcup_{\mathsf{E}} E_2 \sqcup_{\mathsf{E}} [\{v\}/n],\texttt{tt}} ()} \qquad \lfloor\text{PAUSE}\rfloor\ \frac{}{\mathsf{pause} \xrightarrow[S]{\emptyset,\texttt{ff}} ()}$$

$$\lfloor\text{SIG-P}\rfloor\ \frac{e_1 \xrightarrow[S]{E_1,\texttt{tt}} n \quad n \in S \quad e_2 \xrightarrow[S]{E_2,b} e_2'}{\mathsf{present}\ e_1?\,(e_2) : (e_3) \xrightarrow[S]{E,\texttt{ff}} e_2'} \qquad \lfloor\text{SIG-NP}\rfloor\ \frac{e_1 \xrightarrow[S]{E,\texttt{tt}} n \quad n \notin S}{\mathsf{present}\ e_1?\,(e_2) : (e_3) \xrightarrow[S]{E,\texttt{ff}} e_3}$$

$$\lfloor\text{DU-END}\rfloor\ \frac{e_2 \xrightarrow[S]{E_2,\texttt{tt}} n \quad e_1 \xrightarrow[S]{E_1,\texttt{tt}} v}{\mathsf{do}\,(e_1)\,\mathsf{until}\,e_2(x) \to (e_3) \xrightarrow[S]{E_1 \sqcup_{\mathsf{E}} E_2,\texttt{tt}} v} \qquad \lfloor\text{DU-P}\rfloor\ \frac{e_2 \xrightarrow[S]{E_2,\texttt{tt}} n \quad n \in S \quad e_1 \xrightarrow[S]{E_1,\texttt{ff}} e_1'}{\mathsf{do}\,(e_1)\,\mathsf{until}\,e_2(x) \to (e_3) \xrightarrow[S]{E_1 \sqcup_{\mathsf{E}} E_2,\texttt{ff}} e_3\{S^v(n)/x\}}$$

$$\lfloor\text{DU-NP}\rfloor\ \frac{e_2 \xrightarrow[S]{E_2,\texttt{tt}} n \quad n \notin S \quad e_1 \xrightarrow[S]{E_1,\texttt{ff}} e_1'}{\mathsf{do}\ (e_1)\ \mathsf{until}\ e_2(x) \to (e_3) \xrightarrow[S]{E_1 \sqcup_{\mathsf{E}} E_2,\texttt{ff}} \mathsf{do}\ (e_1')\ \mathsf{until}\ e_2(x) \to (e_3)}$$

Fig. 4. Big-step semantics for RML expressions (selection).

neously, then the whole expression terminates. Rule $\lfloor$DU-P$\rfloor$ says that if e_2 transitions to a currently present signal n, then e_3 is executed in the next instant, substituting x with the values gathered in n. Rule $\lfloor$DU-NP$\rfloor$ executes e_1 as long as e_2 does not reduce to a currently present signal. We shall rely on a simple notion of equality:

Definition 9 (Equality with case normalization). *Let $\hookrightarrow_{\mathtt{R}}$ denote the extension of $\equiv_{\mathtt{R}}$ with the axiom* match c_j with $\{c_i \to P_i\}_{i \in I} \hookrightarrow_{\mathtt{R}} P_j$*, where c_j is a constant and $j \in I$.*

RMLq*: ReactiveML with a Queue-based Semantics* We extend RML with an explicit store of queues that keeps the state of the executed program. Unlike signals, the store of queues is preserved throughout time. The syntax of RML is extended with constructs that modify the queues located in the store; the resulting language is called RMLq:

Definition 10 (RMLq**).** RMLq *expressions are obtained by extending the grammar of values in Def. 6 with the following forms:* $v ::= \cdots \mid \mathtt{pop} \mid \mathtt{put} \mid \mathtt{isEmpty}$.

$$\lfloor \textsc{Put-Q} \rfloor \quad \langle \texttt{put}\ q\ v\,;\ \Sigma, q : \widetilde{h} \rangle \xrightarrow[S]{\emptyset,\texttt{tt}} \langle ()\,;\ \Sigma, q : \widetilde{h} \cdot v \rangle$$

$$\lfloor \textsc{Pop-Q} \rfloor \quad \langle \texttt{pop}\ q\,;\ \Sigma, q : v \cdot \widetilde{h} \rangle \xrightarrow[S]{\emptyset,\texttt{tt}} \langle v\,;\ \Sigma, q : \widetilde{h} \rangle$$

$$\lfloor \textsc{NEmpty} \rfloor \quad \langle \texttt{isEmpty}\ q\,;\ \Sigma, q : \widetilde{h} \rangle \xrightarrow[S]{\emptyset,\texttt{tt}} \langle ()\,;\ \Sigma, q : \widetilde{h} \rangle$$

$$\lfloor \textsc{Pop-Q}_\epsilon \rfloor \quad \langle \texttt{pop}\ q\,;\ \Sigma, q : \epsilon \rangle \xrightarrow[S]{\emptyset,\texttt{ff}} \langle \texttt{pop}\ q\,;\ \Sigma, q : \epsilon \rangle$$

$$\lfloor \textsc{Empty} \rfloor \ \langle \texttt{isEmpty}\ q\,;\ \Sigma, q : \epsilon \rangle \xrightarrow[S]{\emptyset,\texttt{ff}} \langle \texttt{isEmpty}\ q\,;\ \Sigma, q : \epsilon \rangle$$

Fig. 5. Big-step semantics for RMLq: Queue-related operations.

The new constructs allow RMLq programs to modify queues, which are ranged over by $q, q', \ldots$. Construct `put` receives a queue and an element as parameters and pushes the element into the end of the queue. Construct `pop` takes a queue and dequeues its first element; if the queue is empty in the current instant the process will block the current thread until an element is obtained. Construct `isEmpty` blocks a thread until the instant in which a queue stops being empty.

The semantics of RMLq includes a *state* $\Sigma, \Sigma' ::= \emptyset \mid \Sigma, q : \widetilde{v}$ (i.e., a possibly empty collection of queues) and *configurations* $K, K' ::= \langle e\,;\ \Sigma \rangle$. The big-step semantics then has transitions of the form $\langle e\,;\ \Sigma \rangle \xrightarrow[S]{E,b} \langle e'\,;\ \Sigma' \rangle$, where S is a signal environment, b is a termination boolean, and E is an event. The corresponding transition system is generated by rules including those in Fig. 5 (see also [7]).

Most transition rules for RMLq are interpreted as for RML; we briefly discuss queue-related rules in Fig. 5. Rule $\lfloor$Put-Q$\rfloor$ pushes an element into a queue and terminates instantaneously. Rule $\lfloor$Pop-Q$\rfloor$ takes the first element from the queue (if not empty) and terminates instantaneously. Rule $\lfloor$NEmpty$\rfloor$ enables `isEmpty` to terminate instantaneously if the queue is not empty. Rule $\lfloor$Pop-Q$_\epsilon\rfloor$ keeps the thread execution stuck for at least one instant if the queue is empty; Rule $\lfloor$Empty$\rfloor$ is similar. We rule out programs with parallel `pop`/`put` operations along the same session in the same instant.

4 Expressiveness Results

We present our main results: correct translations of π into RML and of aπ into RMLq.

The Formal Notion of Encoding We define notions of language, translation, and encoding by adapting those from Gorla's framework for relative expressiveness [9].

Definition 11 (Languages & Translations). *A* language $\mathcal{L}$ *is a tuple* $\langle \mathsf{P}, \rightarrow, \approx \rangle$*, where* P *is a set of processes,* $\rightarrow$ *denotes an operational semantics, and* $\approx$ *is a behavioral equality on* P*. A* translation *from* $\mathcal{L}_s = \langle \mathsf{P}_s, \rightarrow_s, \approx_s \rangle$ *into* $\mathcal{L}_t = \langle \mathsf{P}_t, \rightarrow_t, \approx_t \rangle$ *(each with countably infinite sets of variables* V_s *and* V_t*, respectively) is a pair* $\langle [\![\cdot]\!], \psi_{[\![\cdot]\!]} \rangle$*, where* $[\![\cdot]\!] : \mathsf{P}_s \rightarrow \mathsf{P}_t$ *is a mapping, and* $\psi_{[\![\cdot]\!]} : \mathsf{V}_s \rightarrow \mathsf{V}_t$ *is a renaming policy for* $[\![\cdot]\!]$*.*

We are interested in *encodings*: translations that satisfy certain correctness criteria:

Definition 12 (Encoding). *Let* $\mathcal{L}_s = \langle \mathsf{P}_s, \rightarrow_s, \approx_s \rangle$ *and* $\mathcal{L}_t = \langle \mathsf{P}_t, \rightarrow_t, \approx_t \rangle$ *be languages; also let* $\langle [\![\cdot]\!], \psi_{[\![\cdot]\!]} \rangle$ *be a translation between them (cf. Def. 11). We say that such a translation is an* encoding *if it satisfies the following criteria:*

1. ***Name invariance**: For all $S \in \mathsf{P}_s$ and substitution σ, there exists σ' such that $[\![S\sigma]\!] = [\![S]\!]\sigma'$, with $\psi_{[\![\cdot]\!]}(\sigma(x)) = \sigma'(\psi_{[\![\cdot]\!]}(x))$, for any $x \in \mathsf{V}_s$.*
2. ***Compositionality**: Let $\mathsf{res}_s(\cdot,\cdot)$ and $\mathsf{par}_s(\cdot,\cdot)$ (resp. $\mathsf{res}_t(\cdot,\cdot)$ and $\mathsf{par}_t(\cdot,\cdot)$) denote restriction and parallel composition operators in P_s (resp. P_t). Then, we define: $[\![\mathsf{res}_s(\widetilde{x}, P)]\!] = \mathsf{res}_t(\psi_{[\![\cdot]\!]}(\widetilde{x}), [\![P]\!])$ and $[\![\mathsf{par}_s(P,Q)]\!] = \mathsf{par}_t([\![P]\!], [\![Q]\!])$.*
3. ***Operational correspondence**, i.e., it is sound and complete: (1) **Soundness:** For all $S \in \mathsf{P}_s$, if $S \rightarrow_s S'$, there exists $T \in \mathsf{P}_t$ such that $[\![S]\!] \Longrightarrow_t T$ and $T \approx_t [\![S']\!]$. (2) **Completeness:** For all $S \in \mathsf{P}_s$ and $T \in \mathsf{P}_t$, if $[\![S]\!] \Longrightarrow_t T$, there exists S' such that $S \Longrightarrow_s S'$ and $T \approx_t [\![S']\!]$.*

While name invariance and compositionality are *static correctness criteria*, operational correspondence is a *dynamic correctness criterion*. Notice that our notion of compositionality is less general than that in [9]: this is due to the several important differences in the structure of the languages under comparison (π vs. RML and $\mathsf{a}\pi$ vs. RMLq).

We shall present translations of π into RML and of $\mathsf{a}\pi$ into RMLq, which we will show to be encodings. We instantiate Def. 11 with the following languages:

Definition 13 (Concrete Languages). *We shall consider:*

- *$\mathcal{L}_\pi$ will denote the tuple $\langle \pi, \longrightarrow, \equiv_{\mathtt{S}} \rangle$, where π is as in Def. 1; $\longrightarrow$ is the reduction semantics in Fig. 1; and $\equiv_{\mathtt{S}}$ is the structural congruence relation for π.*
- *$\mathcal{L}_{\mathsf{RML}}$ will denote the tuple $\langle \mathsf{RML}, \xrightarrow[S]{E,b}, \hookrightarrow_{\mathtt{R}} \rangle$, where RML is as in Def. 6; $\xrightarrow[S]{E,b}$ is the big-step semantics for RML; and $\hookrightarrow_{\mathtt{R}}$ is the equivalence in Def. 9.*
- *$\mathcal{L}_{\mathsf{a}\pi}$ will denote the tuple $\langle \mathsf{a}\pi, \longrightarrow_{\mathtt{A}}, \equiv_{\mathtt{A}} \rangle$, where $\mathsf{a}\pi$ is as in Def. 3; $\longrightarrow_{\mathtt{A}}$ is the reduction semantics in Fig. 2; and $\equiv_{\mathtt{A}}$ is the structural congruence relation for $\mathsf{a}\pi$.*
- *$\mathcal{L}_{\mathsf{RMLq}}$ will denote the tuple $\langle \mathsf{RMLq}, \overset{E,b}{\underset{S}{\dashrightarrow}}, \equiv_{\mathtt{R}} \rangle$, where RMLq is as in Def. 10; $\overset{E,b}{\underset{S}{\dashrightarrow}}$ is the big-step semantics for RMLq; and $\equiv_{\mathtt{R}}$ is the equivalence for RML.*

When events, termination booleans, and signal environments are unimportant, we write $P \longmapsto Q$ instead of $P \xrightarrow[S]{E,b} Q$, and $K \mapdashrightarrow K'$ instead of $K \overset{E,b}{\underset{S}{\dashrightarrow}} K'$.

Encoding $\mathcal{L}_\pi$ into $\mathcal{L}_{\mathsf{RML}}$ Key aspects in our translation of $\mathcal{L}_\pi$ into $\mathcal{L}_{\mathsf{RML}}$ are: (i) the use of value carrying signals to model communication channels; and (ii) the use of a continuation-passing style (following [8]) to model variables in π using RML signals.

Definition 14 (Translating $\mathcal{L}_\pi$ into $\mathcal{L}_{\mathsf{RML}}$). *Let $\langle [\![\cdot]\!]_f, \psi_{[\![\cdot]\!]_f} \rangle$ be a translation where: (1) $\psi_{[\![\cdot]\!]_f}(x) = x$, i.e., every variable in π is mapped to the same variable in RML. (2) $[\![\cdot]\!]_f : \mathcal{L}_\pi \rightarrow \mathcal{L}_{\mathsf{RML}}$ is as in Fig. 6, where f is a substitution function.*

Function f in $[\![\cdot]\!]_f$ ensures that fresh signal identifiers are used in each protocol action. The translation of $x\langle v \rangle.P$ declares a new signal x' which will be sent paired with value v through signal x; process $[\![P]\!]_{f,\{x \leftarrow x'\}}$ is executed in the next instant. Dually, the translation of $x(y).P$ awaits a signal carrying a pair, composed of a value and the signal name that to be used in the continuation, which is executed in the next instant. Translations for selection and branching are special cases of those for output and input. Restriction $(\boldsymbol{\nu} xy)P$ is translated by declaring a fresh signal w, which replaces

$$
\begin{aligned}
[\![x\langle v\rangle.P]\!]_f &\triangleq \mathsf{signal}\ x'\ \mathsf{in}\ (\mathsf{emit}\ f_x\ (v,x');\mathsf{pause}\,;[\![P]\!]_{f,\{x\leftarrow x'\}})\\
[\![x(y).P]\!]_f &\triangleq \mathsf{await}\ f_x(y,w)\ \mathsf{in}\ [\![P]\!]_{f,\{x\leftarrow w\}}\\
[\![x \triangleleft l.P]\!]_f &\triangleq \mathsf{signal}\ x'\ \mathsf{in}\ (\mathsf{emit}\ f_x\ (l,x');\mathsf{pause}\,;[\![P]\!]_{f,\{x\leftarrow x'\}})\\
[\![x \triangleright l_j\{l_i : P_i\}_{i\in I}]\!]_f &\triangleq \mathsf{await}\ f_x(l,w)\ \mathsf{in}\ \mathsf{match}\ l\ \mathsf{with}\ \{l_i \to [\![P_i]\!]_{f,\{x\leftarrow w\}}\}\\
[\![v?\,(P)\mathbf{:}(Q)]\!]_f &\triangleq \mathsf{if}\ v\ \mathsf{then}\ (\mathsf{pause}\,;P)\ \mathsf{else}\ (\mathsf{pause}\,;Q)\\
[\![(\boldsymbol{\nu} xy)P]\!]_f &\triangleq \mathsf{signal}\ w\ \mathsf{in}\ [\![P]\!]_{f,\{x\leftarrow w,y\leftarrow w\}}\\
[\![*\,x(y).P]\!]_f &\triangleq \mathsf{let\ rec\ process}\ repl\ \alpha\ \beta =\\
&\qquad \mathsf{signal}\ x'\ \mathsf{in}\\
&\qquad\ \mathsf{do}\ (\mathsf{loop\ present}\ f_\alpha?\,(\mathsf{emit}\ x'\,;\mathsf{pause}\,) : (())\ \mathsf{until}\ f_\alpha(y,w)\\
&\qquad\quad \to (\mathsf{run}\ \beta_{\{\alpha\leftarrow w\}})\\
&\qquad\ \parallel \mathsf{await}\ x'\ \mathsf{in\ run}\ (repl\ \alpha\ \beta)\\
&\qquad \mathsf{in\ run}\ repl\ x\ (\mathsf{process}\ [\![P]\!]_f)\\
[\![P \mid Q]\!]_f &\triangleq [\![P]\!]_f \parallel [\![Q]\!]_f \qquad\qquad [\![\mathbf{0}]\!]_f \triangleq ()
\end{aligned}
$$

Fig. 6. Translation from $\mathcal{L}_\pi$ to $\mathcal{L}_{\mathsf{RML}}$ (Def. 14). Notice that f_x is a shorthand for $f(x)$.

x, y in $[\![P]\!]_f$. Conditionals, parallel composition and inaction are translated homomorphically. Input-guarded replication is a special case of recursion, enabling at most one copy of the spawned process in the same instant; such a copy will be blocked until the process that spawned it interacts with some process. In Fig. 6, α, β denote variables inside the declaration of a recursive process, distinct from any other variables.

We state our first technical result: the translation of $\mathcal{L}_\pi$ into $\mathcal{L}_{\mathsf{RML}}$ is an encoding. In the proof, we identify a class of *well-formed* π processes that have at most one output and selection per endpoint in the same instant; see [7] for details.

Theorem 1. *Translation $\langle [\![\cdot]\!]_f, \psi_{[\![\cdot]\!]_f}\rangle$ is an encoding, in the sense of Def. 12.*

Encoding $\mathcal{L}_{\mathsf{a}\pi}$ into $\mathcal{L}_{\mathsf{RMLq}}$ The main intuition in translating $\mathsf{a}\pi$ into RMLq is to use the queues of RMLq coupled with a *handler process* that implements the output-input transmission between queues. We start by introducing some auxiliary notions.

Notation 1 *Let $P \equiv_{\mathtt{A}} (\boldsymbol{\nu}\widetilde{x})(\prod_{i\in\{1,\ldots,n\}} Q_i \mid \prod_{k_j\in\widetilde{k}} k_j[i:\epsilon, o:\epsilon]$ be PI (cf. Def. 5) with variables $\widetilde{k}$. We will write P as $C_l[Q_l, \mathcal{K}(\widetilde{k})]$, where $l \in \{1,\ldots,n\}$, $C_l[\cdot_1,\cdot_2] = \prod_{j\in\{1,\ldots,n\}\setminus\{l\}} Q_j \mid [\cdot_1] \mid [\cdot_2]$, and $\mathcal{K}(\widetilde{k}) = \prod_{k_j\in\widetilde{k}} k_j[i:\epsilon, o:\epsilon]$.*

This notation allows us to distinguish two parts in a PI process: the non-queue processes and the queue processes $\mathcal{K}(\widetilde{k})$. We now define the key notion of *handler process*:

Definition 15 (Handler process). *Given $\widetilde{k} = \{k_1,\ldots,k_n\}$, the handler process $\mathcal{H}(\widetilde{k})$ is defined as $\prod_{i\in\{1,\ldots,n\}} I(k_i) \parallel O(k_i)$, where $I(k)$ and $O(k)$ are as in Fig. 7.*

Given an endpoint k, a handler defines parallel processes I^k and O^k to handle input and output queues. Transmission is a handshake where both O^k and $I^{\overline{k}}$ (or viceversa) must be ready to communicate. If ready, O^k sends a pair containing the message ($\mathtt{pop}\ k_o$) and a fresh signal for further actions (α'). Once the pair is received, it is enqueued in $\overline{k}_i$ (i.e., the dual $I^{\overline{k}}$). The process is recursively called in the next instant with the new endpoints. The translation of $\mathsf{a}\pi^\star$ into RMLq requires a final auxiliary definition:

$$
\begin{array}{l}
I(k) \triangleq \ \mathsf{let\ rec\ process}\ I\ \alpha = \\
\quad (\mathsf{present}\ ack^{\alpha}?\,(\mathsf{emit}\ ack^{\overline{\alpha}}\,;\mathsf{await}\ \alpha(x,\alpha')\ \mathsf{in}\ (\mathtt{put}\ x\ k_i);\mathsf{run}\ I\ \overline{\alpha'}) : (I\ \alpha) \\
\quad \mathsf{in\ run}\ I\ k \\
O(k) \triangleq \ \mathsf{let\ rec\ process}\ O\ \alpha = \\
\qquad \mathsf{signal}\ \alpha'\ \mathsf{in}\ \mathtt{isEmpty}\ \alpha_o; \mathsf{emit}\ ack^{\overline{\alpha}}\,; \\
\qquad (\mathsf{present}\ ack^{\alpha}?\,(\mathsf{emit}\ \overline{\alpha}\ ((\mathtt{pop}\ k_o),\alpha');\mathsf{pause}\,;\mathsf{run}\ O\ \alpha') : (\mathsf{run}\ O\ \alpha) \\
\quad \mathsf{in\ run}\ O\ k
\end{array}
$$

Fig. 7. Components of handler processes (Def. 15)

$$
\begin{array}{rlrl}
\{\!|x(y).P|\!\} & \triangleq \mathsf{let}\ y = \mathtt{pop}\ x_i\ \mathsf{in}\ \{\!|P|\!\} & \{\!|x\langle v\rangle.P|\!\} & \triangleq \mathtt{put}\ x_o\ v; \{\!|P|\!\} \\
\{\!|x \triangleright \{l_i : P_i\}_{i\in I}|\!\} & \triangleq \mathsf{let}\ y = \mathtt{pop}\ x_i\ \mathsf{in} & \{\!|x \triangleleft l.P|\!\} & \triangleq \mathtt{put}\ x_o\ l; \{\!|P|\!\} \\
 & \quad \mathsf{match}\ l\ \mathsf{with}\ \{l_i : \{\!|P_i|\!\}\}_{i\in I} & \{\!|P \mid Q|\!\} & \triangleq \{\!|P|\!\} \parallel \{\!|Q|\!\} \\
\{\!|b?\,(P)\mathbf{:}(Q)|\!\} & \triangleq \mathsf{if}\ b\ \mathsf{then}\ \{\!|P|\!\}\ \mathsf{else}\ \{\!|Q|\!\} & \{\!|(\boldsymbol{\nu} x)P|\!\} & \triangleq \mathsf{signal}\ x,\overline{x}\ \mathsf{in}\ \{\!|P|\!\} \\
\{\!|\mu X.P|\!\} & \triangleq \mathsf{let\ rec\ process}\ \alpha_X = \{\!|P|\!\}\ \mathsf{in} & \{\!|X|\!\} & \triangleq \mathsf{pause}\,;\mathsf{run}\ \alpha_X \\
 & \quad \mathsf{run}\ \alpha_X & \{\!|\mathbf{0}|\!\} & \triangleq ()
\end{array}
$$

Fig. 8. Auxiliary translation from $\mathsf{a}\pi^{\star}$ into RMLq (Def. 17).

Definition 16. *We define $\delta(\cdot)$ as a function that maps $\mathsf{a}\pi$ processes into* RMLq *states:* $\delta(k[i:\widetilde{h}; o:\widetilde{m}]) = \{k_i:\widetilde{h}, k_o:\widetilde{m}\}$ $\quad \delta(P \mid Q) = \delta(P) \cup \delta(Q)$ $\quad \delta((\boldsymbol{\nu}x)P) = \delta(P)$ *and as $\delta(P) = \emptyset$ for every other $\mathsf{a}\pi$ process.*

Definition 17 (Translating $\mathcal{L}_{\mathsf{a}\pi}$ into $\mathcal{L}_{\mathsf{RMLq}}$). *Let $\langle (\!|\cdot|\!), \psi_{(\!|\cdot|\!)}\rangle$ be a translation where:*

- *$\psi_{(\!|\cdot|\!)}(k) = k$, i.e., every variable in $\mathsf{a}\pi$ is mapped to the same variable in* RMLq.
- *$(\!|\cdot|\!) : \mathcal{L}_{\mathsf{a}\pi} \to \mathcal{L}_{\mathsf{RMLq}}$ is defined for properly initialized $\mathsf{a}\pi$ processes $C[Q, \mathcal{K}(\widetilde{k})]$, which are translated into* RMLq *configurations as follows:*

$$(\!|C[Q,\mathcal{K}(\widetilde{k})]|\!) = \langle \{\!|C[Q,\mathbf{0}]|\!\} \parallel \mathcal{H}(\widetilde{k})\,;\, \delta(\mathcal{K}(\widetilde{k}))\rangle$$

where $\{\!|\cdot|\!\} : \mathcal{L}_{\mathsf{a}\pi^{\star}} \to \mathcal{L}_{\mathsf{RMLq}}$ is in Fig. 8; $\mathcal{H}(\widetilde{k})$ is in Def. 15; and $\delta(\cdot)$ is in Def. 16.

Two key ideas in translation $(\!|\cdot|\!)$ are: *queues local to processes* and *compositional (queue) handlers*. Indeed, communication between an endpoint k and its queues k_i, k_o proceeds instantaneously, for such queues should be local to the process implementing session k. Queue handlers effectively separate processes/behavior from data/state. As such, it is conceivable to have handlers that have more functionalities than those of $\mathcal{H}(\widetilde{k})$. In [7] we provide an example of a handler more sophisticated than $\mathcal{H}(\widetilde{k})$.

Translation $(\!|\cdot|\!)$ is in two parts. First, $\{\!|\cdot|\!\}$ translates non-queue processes: output and input are translated into queuing and dequeuing operations, respectively. Selection and branching are modeled similarly. Translations for the conditional, inaction, parallel, and recursion is as expected. Recursion is limited to a pause-guarded tail recursion in $\{\!|\cdot|\!\}$ to avoid loops of instantaneous expressions and nondeterminism when accessing queues. Second, $(\!|\cdot|\!)$ creates an RML configuration by composing the RMLq process obtained via $\{\!|\cdot|\!\}$ with appropriate handlers and with the state obtained from the information in $\mathsf{a}\pi$

queues. Because of this two-part structure, static correctness properties are established for $\{\!|\cdot|\!\}$ (for this is the actual translation of source processes), whereas operational correspondence is established for $(\!|\cdot|\!)$ (which generates an RMLq configuration).

Theorem 2 (Name invariance and compositionality for $\{\!|\cdot|\!\}$). *Let P, σ, x, and $E[\cdot]$ be an $\mathtt{a}\pi^\star$ process, a substitution, a variable in $\mathtt{a}\pi^\star$, and an evaluation context (cf. Def. 4), respectively. Then: (1) $\{\!|P\sigma|\!\} = \{\!|P|\!\}\sigma$, and (2) $\{\!|E[P]|\!\} = \{\!|E|\!\}\big[\{\!|P|\!\}\big]$.*

Theorem 3 (Operational correspondence for $(\!|\cdot|\!)$). *Given a properly initialized $\mathtt{a}\pi$ process $C[Q, \mathcal{K}(\widetilde{k})]$, it holds that:*

1. **Soundness:** *If $C[Q, \mathcal{K}(\widetilde{k})] \longrightarrow_{\mathtt{A}} C[Q', \mathcal{K}'(\widetilde{k})]$ then*
   $(\!|C[Q, \mathcal{K}(\widetilde{k})]|\!) \longmapsto (\!|C'[Q'', \mathcal{K}''(\widetilde{k})]|\!)$, *for some $Q'', \mathcal{K}''(\widetilde{x}), C'$ where $C[Q, \mathcal{K}(\widetilde{x})] \longrightarrow_{\mathtt{A}} C[Q', \mathcal{K}'(\widetilde{x})] \longrightarrow^*_{\mathtt{A}} (\boldsymbol{\nu}\widetilde{x})C'[Q'', \mathcal{K}''(\widetilde{x})]$.*
2. **Completeness:** *If $(\!|C[Q, \mathcal{K}(\widetilde{x})]|\!) \longmapsto R$ then there exist Q',C',$\mathcal{K}'(\widetilde{x})$ such that $C[Q, \mathcal{K}(\widetilde{x})] \longrightarrow^*_{\mathtt{A}} (\boldsymbol{\nu}\widetilde{x})C'[Q', \mathcal{K}'(\widetilde{x})]$ and $R = (\!|C'[Q', \mathcal{K}'(\widetilde{x})]|\!)$.*

In soundness, a single RMLq step mimicks one or more steps in $\mathtt{a}\pi$, i.e., several source computations can be grouped into the same instant. This way, e.g., the interaction of several outputs along the same session with their queue (cf. Rule $\lfloor$SEND$\rfloor$) will take place in the same instant. In contrast, several queue synchronizations in the same session (cf. Rule $\lfloor$COM$\rfloor$) will be sliced over different instants. Conversely, completeness ensures that our encoding does not introduce extraneous behaviors: for every RMLq transition of a translated process there exists one or more corresponding $\mathtt{a}\pi$ reductions.

5 Closing Remarks

We have shown that ReactiveML can correctly encode session-based concurrency, covering both synchronous and asynchronous (queue-based) communications.[4] Our encodings are *executable*: as such, they enable to integrate session-based concurrency in actual RML programs featuring declarative, reactive, timed, and contextual behavior. This is an improvement with respect to previous works, which extend the π-calculus with some (but not all) of these features and/or lack programming support. Interestingly, since ReactiveML has a well-defined semantics, it already offers a firm basis for both foundational and practical studies on session-based concurrency. Indeed, ongoing work concerns the principled extension of our approach to the case of multiparty sessions.

We have not considered types in source/target languages, but we do not foresee major obstacles. In fact, we have already shown that our encoding $[\![\cdot]\!]_f$ supports a large class of well-typed π processes in the system of [18], covering a typed form of operational correspondence but also *type soundness*: if P is a well-typed π process, then $[\![P]\!]_f$ is a well-typed RML expression—see [7]. We conjecture a similar result for $(\!|\cdot|\!)$, under an extension of [18] with queues. On the ReactiveML side, we can exploit the type-and-effect system in [14] to enforce *cooperative* programs (roughly, programs without infinite loops). Since $[\![\cdot]\!]_f$ and $(\!|\cdot|\!)$ already produce well-typed, executable ReactiveML expressions, we further conjecture that they are also cooperative, in the sense of [14].

[4] *Synchronous communication* as in the (session) π-calculus should not be confused with the *synchronous programming* model of ReactiveML.

Acknowledgements We thank Ilaria Castellani, Cinzia Di Giusto, and the anonymous reviewers for useful remarks and suggestions. This work has been partially sponsored by CNRS PICS project 07313 (SuCCeSS) and EU COST Actions IC1201 (BETTY), IC1402 (ARVI), and IC1405 (Reversible Computation).

References

1. M. Bartoletti, T. Cimoli, M. Murgia, A. S. Podda, and L. Pompianu. Compliance and subtyping in timed session types. In *FORTE*, volume 9039 of *LNCS*, pages 161–177. Springer, 2015.
2. A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.
3. L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR 2010*, volume 6269 of *LNCS*, pages 162–176. Springer - Verlag, 2010.
4. L. Bocchi, W. Yang, and N. Yoshida. Timed multiparty session types. In *Proc. of CONCUR'14*, volume 8704, pages 419–434. Springer, 2014.
5. E. Bonelli, A. B. Compagnoni, and E. L. Gunter. Correspondence assertions for process synchronization in concurrent communications. *J. Funct. Program.*, 15(2):219–247, 2005.
6. F. Boussinot and R. de Simone. The SL synchronous language. *IEEE Trans. Software Eng.*, 22(4):256–266, 1996.
7. M. Cano, J. Arias, and J. A. Pérez. Session-based Concurrency, Reactively (Extended Version), 2017. Available at `http://www.jperez.nl/publications`.
8. O. Dardha, E. Giachino, and D. Sangiorgi. Session types revisited. In *Proc. of PPDP'12*, pages 139–150, 2012.
9. D. Gorla. Towards a unified approach to encodability and separation results for process calculi. *Inf. Comput.*, 208(9):1031–1053, 2010.
10. N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Trans. Software Eng.*, 18(9):785–793, 1992.
11. K. Honda, V. T. Vasconcelos, and M. Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *Proc. of ESOP'98*, volume 1381, pages 122–138. Springer, 1998.
12. H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, P.-M. Deniélou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, H. T. Vieira, and G. Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, Apr. 2016.
13. D. Kouzapas, N. Yoshida, R. Hu, and K. Honda. On asynchronous eventful session semantics. *Mathematical Structures in Computer Science*, 26(2):303–364, 2016.
14. L. Mandel and C. Pasteur. Reactivity of Cooperative Systems - Application to ReactiveML. In *21st International Symposium, SAS 2014, Munich, Germany, 2014.*, pages 219–236, 2014.
15. L. Mandel, C. Pasteur, and M. Pouzet. ReactiveML, ten years later. In M. Falaschi and E. Albert, editors, *Proc. of PPDP 2015*, pages 6–17. ACM, 2015.
16. L. Mandel and M. Pouzet. ReactiveML: a reactive extension to ML. In *Proc. of PPDP'05*, pages 82–93. ACM, 2005.
17. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992.
18. V. T. Vasconcelos. Fundamentals of session types. *Inf. Comput.*, 217:52–70, 2012.